

-1-

Date: May 25, 2000 Express Mail Label No. EL564266910US

Inventor: James E. Carey

Attorney's Docket No.: 1958.2001-000

PARALLEL TASK SCHEDULING SYSTEM FOR COMPUTERS

BACKGROUND

In computers, application programs execute software instructions on a processor to perform work. Modern computers allow those instructions to be divided into discrete tasks for processing. In a multi-threaded computing environment, the tasks are assigned to multiple computing threads for processing. The threads perform the task and return results to the application program.

In a typical free-thread environment, any available thread can be used to process a task. Tasks are assigned to worker threads by provider threads executing thread manager instructions. There is no predefined relationship between a worker thread and a task or application.

Typically, the thread manager queues tasks into a single task queue. When a worker thread becomes available, the next task in the task queue is assigned to that worker thread on a first-in, first-out basis. On a busy system, the worker threads can all be busy at any particular time. As a result, new tasks cannot be immediately assigned to a thread. This causes the single queue to become populated with waiting tasks.

SUMMARY

One problem with using a single queue to feed tasks to a plurality of worker threads is that it results in lock contention. When a worker thread becomes available, the queue is locked until the first waiting task is located and assigned to the thread. Subsequently freed threads must wait on that lock before proceeding, causing a

bottleneck. In a busy system, the free - but idling - worker threads can become a wasted resource because they spend a considerable amount of time in a wait state.

One solution to the problem is to define a task space as a plurality of task queues. Each task queue can then be associated with a respective worker thread. This approach can diminish lock contention problems because a free worker thread would generally only cause its own task queue to be locked. Other subsequently freed worker threads could continue to process tasks from their own task queues.

Such parallel task queues can use a task scheduling algorithm to distribute tasks amongst the queues. To obtain an even distribution, a random number generator can be employed to select a task queue. Although the randomly selected queue may be busy, it provides a starting point for locating an empty queue. Once an empty queue is located, the new task is placed on that queue for processing by the associated task.

While the randomization can evenly distribute the work, the task still may not be efficiently removed from its assigned queue. To reduce the waiting time of queued tasks, the task scheduling algorithm can include a method of stealing a queued task. In particular, a freed worker thread first checks its associated queue. If the queue is empty, the worker thread searches the other queues for a waiting task. That task can then be moved to the empty queue and processed by the worker thread.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features and advantages of the parallel task scheduling system for computers will be apparent from the following more particular description of embodiments, as illustrated in the accompanying drawings in which like reference characters refer to the same parts throughout the different views. The drawings are not necessarily to scale, emphasis instead being placed upon illustrating the principles of the invention.

FIG. 1 is a schematic block diagram of a client-server computing environment.

FIG. 2 is a block diagram of a prior art task scheduling system.

FIG. 3 is a schematic block diagram of a parallel task scheduling system.

FIG. 4 is a flowchart for a queue assignment method of the parallel task scheduler of FIG. 3.

FIG. 5 is a schematic block diagram of the system of FIG. 3 showing the assignment of a new task.

5 FIG. 6 is a flowchart of a task stealing method of the parallel task scheduler of FIG. 3.

FIG. 7 is a schematic diagram of the system of FIG. 3 showing the assignment of a stolen task.

DETAILED DESCRIPTION

10 FIG. 1 is a schematic block diagram of a client-server computing environment. In the environment 1, a client computer 10 communicates with a server computer 20 over a communications medium 5. The communications medium 5 may be any wired or wireless interconnection between the client 10 and server 20, including a direct network connection or a public switched communications network, such as the Internet.

15 The server 20 can be a multi-processor 22 computer for accessing and manipulating data stored in a data store 29.

As shown, the client 10 is generally a single-processor 12 computer under the control of an operating system. The client 10 executes client application software 18 to perform work for a user. Some of that work may need to be handled by the server 20.

20 In that event, a request is passed by the client 10 to the server 20.

The server 20 receives the request for processing by a server application 28. The software instructions that process that request have been assigned by the compiler to one or more discrete tasks. A task scheduling system 26 within the server operating system 24 is responsible for making sure each task is processed. After each task is

25 completed, results are returned to the server application 28. Ultimately, the client request is filled and results are returned to the client application 18.



5

10

20

25

serially access the queue 40'. This single-threaded access can cause a large amount of thread context switching and can be very inefficient.

FIG. 3 is a schematic block diagram of a parallel task scheduling system 26. As in FIG. 2, a plurality of worker threads $W1...Wx$ are maintained in a worker thread pool 30. There are, however, a plurality of task queues $Q1,...,Qx$ in a queue space 40. As illustrated, each task queue $Q1...Qx$ is associated with a respective worker thread $W1,...,Wx$. Each task queue can store a plurality of individual tasks. A parallel task scheduler 25 manages the assignment of tasks to threads.

As illustrated, the first task queue $Q1$ has one queued task, $T1$. The second task queue $Q2$ is empty. The last task queue Qx has two queued tasks $T2, Tm$. Here, although the second worker thread $W2$ is free, the last worker thread Wx is overburdened.

Such bottlenecks can occur because not all tasks are of equal complexity. A worker thread that draws a complex task can have a populated task queue, while a worker thread that draws simple tasks has an empty task queue. As a result, the processing of a simple task can be delayed by a complex task. It is therefore still possible that the associated worker thread (e.g. Wx) is overburdened by a complex task (e.g. $T2$) and will not process the queued task (e.g. Tm) immediately.

To reduce bottlenecks, the task scheduling system 26 attempts to place a new task on an empty task queue, if one exists. In particular, the task scheduling algorithm uses a random number generator to identify an initial, seed queue. If that randomly selected queue is not empty, the algorithm marches through the queues - starting from the seed queue - until an empty queue is found. The task is then placed on that queue.

Just because an empty queue has been found, however, does not guarantee that the queued task will be processed quickly. The associated worker thread may still be busy processing a complex task. The queue task may have to wait for the processing task to finish. Also, depending on the implementation of the system 26 and the system configuration, the randomly selected queue may not be empty.

Because any worker thread is suitable for processing any task, the parallel task scheduling system 26 can take advantage of additional methods to increase performance. In particular, another method is used by freed worker threads to steal waiting tasks from busy queues.

- 5 Using the parallel queue approach, each task queue is primarily processed by the associated (or owning) worker thread, with periodic access from the task provider thread and infrequent access from the other worker threads as they finish their tasks. Because there are an increased number of locks controlling access to the queues and a decreased number of threads attempting to gain access to the queues, the process is much more
10 efficient and scalable.

FIG. 4 is a flowchart for a queue assignment method of the parallel task scheduler of FIG. 3. The queue assignment method 50 addresses how new tasks are queued for processing. This method is executed by a provider thread P.

- The method first selects a random queue at step 51. A pseudo-random number
15 generator (PRNG) is used to pick a queue number, modulo x , when x is the count of task queues. It should be noted that the particular PRNG is not critical to operation of the method, so any convenient PRNG can be used.

- At step 52, the selected queue is locked by the method and first examined to determine if it is an empty queue. If the queue is not empty, the lock may be released
20 and the next queue selected at step 53 by incrementing the queue number (modulo x). This process can continue until an empty queue is found, at step 52. In addition, the search can be halted after a specific time-out condition has been met, such as a predetermined number of increments. In an appropriately configured system, however, an empty queue should be found with little searching.

- 25 In another embodiment of step 52, the provider thread “peeks” at the associated queue, without a lock, to see if the queue is in a busy state. This is done by looking at the queue without holding the Mutex. While the peeker cannot operate on the queue using the information retrieved from the peek, the answer to the question “Is the queue empty?” is valid. This check, while not guaranteed accurate, can be very fast and

allows worker threads that may be busy to be skipped with little penalty to the speed of the check.

The protocol described for the queuing task guarantees that a task deposited by the task provider will have an active thread if the queue belonged to a waiting thread. If the queue is not empty, then the matching worker must either be busy, or be about to remove the task. If there is a task queued, the Mutex is taken to be sure it is really there (not in the process of being removed); if there is no task there, we do not need to take the lock to be sure there is no task there.

Consequently, when a queue is found that does not appear busy, the controlling lock on the queue is taken. If the queue is really empty, the task is deposited on the associated queue. In a small number of cases, the queue may no longer be empty after the lock is taken. In that case, the lock is dropped and the search continues. It is important to note that this type of collision should happen infrequently - e.g., on a very busy server.

In any event, an empty queue will generally be found. At step 54, the task is placed on the selected queue and the method releases the lock. The worker thread associated with that queue should process the task. If the associated worker thread is busy processing a complex task, it may take a relatively long time for the worker thread to again access its queue. Unless dealt with, that scenario could reduce processing efficiency.

FIG. 5 is a schematic block diagram of the system of FIG. 3 showing the assignment of a new task. As shown, the task assignment method 40 (FIG. 4) has found the empty task queue Q2 (FIG. 3). As a result, the new task Tn has been added to that queue Q2 for processing by the associated worker thread W2.

FIG. 6 is a flowchart of a task stealing method of the parallel task scheduler of FIG. 3. The method 60 is initiated by a worker thread completing a task. In a particular embodiment, upon completing a task, a worker thread goes through a wake-up process to reinitialize its thread data and to grab a new task. One problem is that the associated queue can be empty, while other queues are populated. In general, such a situation may

arise only when there are more tasks available for processing than there are threads to process the tasks.

At step 61, the worker queue is examined. If the queue is populated with a task, then processing jumps to step 66 to process that task. If the queue is empty, however,
 5 the method 60 begins searching for a task queued in a queue for another worker thread.

The effort of finding another task begins at step 62, where, the next queue is selected. The selection can be made by simply incrementing the queue number, modulo x. Other techniques can also be used.

Processing then continues to step 63, where the selected queue is examined. If a
 10 task is queued, processing continues to step 65. Otherwise, processing continues to step 64.

At step 64, an optional time-out check can be made. In one embodiment, the check is based on a complete cycle through the queues. That is, if the queue number is incremented back to the worker's queue number, then processing can jump to step 67 to
 15 discontinue. The time-out could also be a predetermined number of increments. The time-out could also result from an interrupt resulting from a task being queued to the worker thread's previously empty queue. As another alternative, idle threads can continuously scan for stealable tasks. Until a time-out, processing returns to step 62 to select the next candidate queue.

20 Once a stealable task is found, the task is moved from the selected queue to the worker's queue at step 65. At step 66, the first task in the worker's queue is processed by the worker thread.

After completion of the task, or a timeout, the worker thread is placed into a sleep mode at step 67.

25 FIG. 7 is a schematic diagram of the system of FIG. 3 showing the assignment of a stolen task. As shown, the task stealing method 50 (FIG. 6) has been used by the free worker thread W2 to steal the waiting task Tm from the busy queue Qx. The stolen task Tm is now in the second queue Q2 for processing by the associated worker thread W2. That task Tm can now be more efficiently handled.

Those of ordinary skill in the art will recognize that methods involved in the parallel task scheduling system for computers may be embodied in a computer program product that includes a computer usable medium. For example, such a computer usable medium can include a readable memory device, such as a solid state memory device, a
5 hard drive device, a CD-ROM, a DVD-ROM, or a computer diskette, having computer readable program code segments stored thereon. The computer readable medium can also include a communications or transmission medium, such as a bus or a communications link, either optical, wired, or wireless, having program code segments carried thereon as digital or analog data signals.

10 While this parallel task scheduling system for computers has been particularly shown and described with references to particular embodiments thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the scope of the invention encompassed by the appended claims. For example, the methods of the invention can be applied to various
15 environments, and are not limited to the environment described herein.